CAMPUS
DE EXCELENCIA
INTERNACIONAL

Trabajo de fin de grado

Diseño y desarrollo de un prototipo de solución de backoffice B2C para bienes digitales sobre Heroku PaaS

Memoria final

June 9, 2013

Tutor:
Javier Soriano

Autor:
Salvador Pérez Martín
02724676G
R090089

Resumen

Dentro del mundo empresarial se denomina *backoffice* a todo lo que ocurre en la empresa que no guarda una relación directa con el cliente. Si concretamos el concepto en el ámbito de una empresa de bienes digitales, éste se reduce al cobro de dichos bienes. Por otro lado, en los últimos años se ha producido un gran aumento del interés en torno al desarrollo de aplicaciones en cloud.

En este contexto, el objetivo de este trabajo es desarrollar una prueba de concepto sobre una plataforma de *backoffice* para bienes digitales en cloud. A través de la realización de esta prueba de concepto se han puesto a prueba las supuestas facilidades de estas plataformas cloud para el desarrollo de aplicaciones, y finalmente se ha analizado el ahorro en el time-to-market que se consigue al utilizarlas.

Los resultados obtenidos han sido positivos, ya que la solución de *backoffice* ha quedado desplegada en cloud y se ha determinado un gran ahorro en el time-to-market gracias al uso de una plataforma PaaS como Heroku y las facilidades que ésta ofrece.

Abstract

Inside business world, *backoffice* is known to be everything that happens inside a company that has noothing to do with the customer directly. If we focus on a Digital goods company, the *backoffice* concept gets reduced to charge those goods. On the other hand, These last years Cloud app develop's hype has increased dramatically.

That is the reason why this assignment goals are to develop a proof of concept of a digital goods' *backoffice* platform on Heroku. We have also tested the facilities that using these cloud platforms to develop applications is supposed to be. Finally and we have made an assessment about the time to market that we save when using these cloud platforms to develop applications.

The results that we have obtained are very positive, because the *backoffice* solution has been successfully deployed on a cloud and we have spotted a great saving on the time to market when using a cloud platform.

Agradecimientos

Antes de nada, me gustaría dedicar un breve espacio para poder agradecer a toda la gente que, de una forma u otra, me han ayudado en la realización de este trabajo.

En primer lugar, me gustaría agradecer a mi tutor, Javier Soriano, la posibilidad de entrar a formar parte, primero del laboratorio, y después de este proyecto que, creo, me ha hecho crecer profesionalmente.

En segundo lugar, quisiera agradecer a mis responsables dentro de telefónica digital, Miguel Ángel Cañas y Agustín Martín, todo el apoyo que me han dado para entender el alcance del proyecto y conseguir tener en mente en todo momento la *"Big picture"* y no limitarme a la parte que teníamos que implementar, así como el soporte que he recibido de su parte para tener todo funcionando.

En tercer lugar a todas las personas que han ofrecido su ayuda en algún momento, ya sea técnica o moral. Aquí entráis mis amigos (Álvaro, Irene, Ley, Marta, Rodrigo, Criss, Yeray y Javi), Mónica, mi hermana y mis compañeros del OIL (Alberto, CD y Sonia).

Finalmente, quisiera agradecérselo a mis padres, por haber estado desde siempre cuidando de que no nos faltara nada ni a mi hermana ni a mí y poniendo nuestra educación por delante de todo.

Muchas gracias.

Contenidos

Lista de figuras

1 Introducción y objetivos

Este Trabajo de Fin de Grado (TFG) se ha realizado en el marco del Laboratorio de innovación abierta UPM - Telefónica digital y ha abordado la implementación de una solución de *backoffice* Business to Consumer (B2C) similar a las usadas por grandes multinacionales de bienes digitales que además podrá ser desplegada en la nube.

El *backoffice* es todo lo que ocurre en una empresa y que no resulta en la interactuación directa con el cliente. Algunos ejemplos de procesos de *backoffice* pueden ser el control de inventario, el abastecimiento de materias primas o el proceso de fabricación. Esto en el ámbito de los bienes digitales se reduce al cobro de los mismos de la manera más transparente posible para el cliente.

Dentro del concepto "cliente" se pueden encontrar dos opciones: Que el cliente sea una empresa o que el cliente sea un usuario final. Si nos encontramos en el primer caso, nos encontraríamos ante una plataforma Business to Business (B2B). El propósito de este TFG, como dijimos antes es implementar el segundo caso, en el que el cliente es un usuario final, Business to Consumer (B2C).

1.1 Motivaciones

La motivación principal de este TFG viene dada principalmente por un problema de agilidad a la hora de desarrollar servicios nuevos dentro de Telefónica. En su estado actual, el protocolo seguido por los desarrolladores consiste en que cada servicio debe implementar una API que será invocada por el servidor de cobros a la hora de facturar a los usuarios de cada servicio. Dicha API debe devolver la cantidad que ha de ser cobrada a cada usuario por el uso de ese servicio concreto.

El problema de esta situación tiene tres vertientes:

- Por un lado, el desarrollo de esta API, que en algunos casos será igual que otras ya implementadas y en otros no, consume mucho tiempo de desarrollo, lo que aumenta considerablemente el time-to-market de dichos servicios.

- Por otro lado, a la hora de realizar dichos cobros se produce una sobrecarga de trabajo en los servidores, ya que cada servicio tiene que calcular por su cuenta la cantidad a cobrar a cada usuario por el uso de ese servicio en concreto, y el servidor central recoger todos estos datos, agruparlos por usuario y lanzar todas las peticiones de cobro.

- Hay que destacar que la sobrecarga de trabajo de la que hablamos en el punto anterior es acrecentada por la forma de almacenamiento de los datos de uso de cada usuario, ya que se hace usando un sistema denominado "service data

record" (sdr), cuyo formato (bastante ineficiente al ser parseado) puede verse en el Anexo A. Cabe destacar que este formato está heredado del usado por telefónica para guardar registro de todas las llamadas telefónicas que se producen para así poder cobrarlas. El nombre de dicho formato es Call Data Record (CDR)

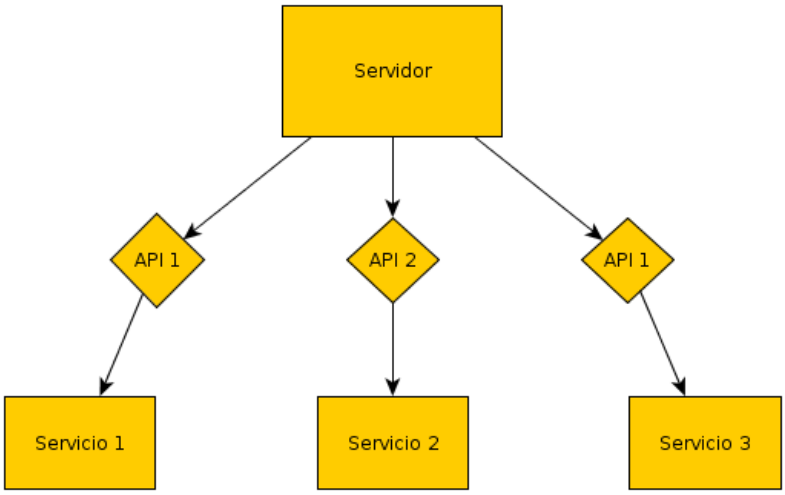


Figura 1: Diagrama de la arquitectura legada

En este contexto surge la necesidad de buscar formas de hacer que el desarrollo de servicios y su puesta en producción sea lo más ágil posible, así como de disminuir la carga de los servidores. Ésto se traduce en dos soluciones que se van a probar juntas:

1. Implementación de una solución de *backoffice* común a todos los servicios que se encargue de los cobros, con lo que los desarrolladores de servicios, lejos de tener que implementar una API entera, sólo necesitarían consumir la API de la que les provee esta solución. Esta solución además disminuirá la carga puntual de trabajo distribuyéndola a lo largo del tiempo.

2. El uso sistemático de plataformas developer-friendly para agilizar tanto la integración del servicio con las tecnologías externas que puedan ser usadas (base de datos, monitorización del rendimiento, sistemas de logging, motores de búsqueda, etc) como para agilizar el despliegue de la aplicación que se esté desarrollando.

1.2 Objetivos

La lista concreta de objetivos a cumplir para este TFG es la siguiente:

- Aprender el uso del PaaS de Heroku, así como la estructura que sigue a la hora de ofrecer sus *addons*.
- Adquirir conocimiento sobre el proyecto *backoffice process manager* a fin de poder continuar con su implementación.
- Integrar la segunda pasarela de pagos (Adyen) para dar soporte a cobros a clientes de servicios alojados en Brasil.
- Diseño del modelo de datos para dotar de robustez al servicio.
- Implementación del nuevo módulo (*Mediator*)
- Estudio de los *addons* de Heroku y su capacidad para producir un ahorro en el *time to market*

1.3 Organización del resto del documento

En esta sección se dará una breve pincelada del contenido del resto de capítulos de este documento.

En el capítulo denominado "Vigilancia tecnológica" se verá que, acorde con el espíritu del área de telefónica digital, el tipo de solución que se ha desarrollado es un ejercicio de innovación, ya que aunque sí existen soluciones de *backoffice*, no existe ninguna que esté desplegada en la nube ni que pueda soportar una gran carga, como va a soportar éste. También se analizarán' todas las opciones que se presentan a la hora de elegir las tecnologías que se han usado durante el desarrollo y se da'rán las causas razonadas de dicha elección. En este ámbito, se ha elegido un framework de desarrollo web (Django) y un proveedor de Cloud (Heroku PaaS). Finalmente, termino dando una visión del estado de evolución de los llamados Heroku addons, que son, como se verá después, soluciones software ya desplegadas en Heroku que no necesitan ser instaladas ni configuradas.

En el capítulo "Desarrollo" se explicará todo el trabajo llevado a cabo, tanto conceptual y arquitectónicamente como programáticamente en los dos repositorios de que consta el proyecto (*backoffice-process-manager* y *Mediator*). Se hablará tanto del modelo arquitectónico seguido como del modelo de datos que se ha diseado para evitar la pérdida de datos.

En el capítulo "Conclusiones" se mencionarán los puntos clave que hay que extraer del proyecto y qué cosas del mismo han de conservarse de cara a próximos trabajos de la misma envergadura (tales como metodologías en el uso de alguna herramienta entre otras).

En el capítulo "Líneas futuras" se expondrán los 3 puntos clave principales de cara a la continuación del proyecto y su paso a producción en el seno de una empresa tan grande como Telefónica, así como tecnologías de las que hay que seguir su evolución para saber si son interesantes de cara a su implantación en el plan tecnológico de telefónica digital.

2 Vigilancia tecnológica

En este capítulo se va a hacer un análisis de las distintas tecnologías entre las que se ha tenido que elegir durante el desarrollo de este tfg para analizar el nivel de desarrollo de cada una y el nivel de uso que han alcanzado hasta el momento. El capítulo está dividido en cinco secciones, una por cada tipo de tecnología.

2.1 Frameworks de desarrollo web

En cuanto al framework usado para crear el servicio, actualmente existen dos tipos entre los que hay que elegir primeramente: los frameworks del lado del cliente y los frameworks del lado del servidor.

Los primeros delegan en el cliente toda la operación de renderizado de la página y todas las operaciones que ello requiere. Si durante estos procesos el cliente necesita datos almacenados en el servidor (cosa que presumiblemente será necesario en la mayor parte de los casos), se pedirn mediante peticiones AJAX. Por tanto, el servidor queda relegado a un mero contenedor de datos. Las ventajas que proporcionan estos frameworks están centradas fundamentalmente en la experiencia de uso (UX), ya que el modo en que funcionan permite que el usuario pueda ver "resultados" mucho antes que con los frameworks tradicionales (centrados en el servidor) y que la página se rellene dinámicamente según se van obteniendo los datos del servidor, permitiendo también modificarla dinámicamente, por lo que el usuario no tendrá que esperar a que la página se recargue con cada acción. En cuanto al servidor, la principal ventaja proporcionada por estos frameworks es que aumenta su capacidad de escalado, ya que al no tener que procesar la información de ninguna manera, sino enviarla en crudo, es capaz de procesar mayor número de peticiones en el mismo periodo de tiempo. Sin embargo, su desventaja también es esa, ya que al tener que pedir al servidor cada dato que se necesite, la red se ve saturada con más paquetes, con lo que puede haber una mayor latencia. El framework más usado de este tipo es Backbone.js[1].

En cuanto a los frameworks del lado del servidor, su funcionamento pasa por delegar en el servidor el renderizado de la página junto con el tratamiento de los datos y el manejo de la base de datos. Las ventajas de este enfoque son la mayor facilidad a la hora de programar, ya que no se depende de los resultados de peticiones web, sino que se realiza directamente en local (evitando en parte tener que lidiar con la asincronía propia de javascript). Las desventajas asociadas a este tipo de tecnología son el retardo que se puede producir en el cliente a la hora de mostrarles la página en cuestión y el tener que recargar con cada acción. No considero como desventaja la disminución de escalabilidad, ya que gracias a la existencia de otras tecnologías que se comentarán después, las task queues, se puede conseguir aumentar dicha

característica. Los dos frameworks más utilizados de los de este tipo actualmente son symfony (escrito en php) y Django (escrito en python).

Para este trabajo nos hemos decantado por el uso de django, el framework escrito en python. Los motivos fundamentales de esta elección son los siguientes:

- El escaso impacto que tendrían en este proyecto las ventajas de los frameworks del lado del cliente (por lo que decidimos usar uno de los dos frameworks del lado del servidor). Esto se debe a que en el caso de nuestra aplicación la mayor parte de las operaciones son aquellas que no tienen que ver con el cliente(Por ejemplo el proceso de cobro).

- La agilidad con la que se puede desarrollar en python. No tenemos datos objetivos que midan esta característica, ya que es un terreno esencialmente subjetivo, pero lo cierto es que durante el desarrollo se tiene en todo momento la sensación de avanzar mucho más rápido que con otros lenguajes de programación tales como java. Esta visión está apoyada por telefónica digital, ya que lo han incluido en su plan tecnológico por idéntico motivo.

- La escasa curva de aprendizaje que tiene python.

- Ha demostrado ser el framework que soporta mayor cantidad de peticiones sin consumir apenas recursos de la máquina[2] (Como vemos en el anexo B).

- Uso del patrón de diseño llamado modelo vista controlador (MVC), creado para facilitar el desarrollo de las aplicaciones.

2.1.1 Modelo Vista Controlador

Como decíamos al final del apartado anterior, el patrón de diseño MVC tiene como finalidad facilitar el desarrollo de todo tipo de aplicaciones mediante la separación del código en tres entidades interrelacionadas (Ver figura 2):

- Modelo: Dentro de este tipo de entidad se encuentra englobado todo el código destinado a tratar con el modelo de datos. Esto incluye definición del modelo, consulta del mismo y operaciones que permitan agregar, modificar y eliminar datos del mismo.

- Vista: Este tipo de entidad engloba todo el código cuya finalidad sea presentar la información al usuario y recoger los datos introducidos al sistema por el mismo. Esto último se refiere solo a la acción de obtenerlos, ya que el tratamiento de los datos lo llevará a cabo el controlador, como veremos ahora.

- Controlador: El controlador es la entidad que conecta el modelo y la vista. Su función consiste en tratar los datos que entran en el sistema, ejecutar las acciones que haya solicitado el cliente (por medio de la vista correspondiente), y tratar los datos de salida a fin de que la vista pueda ser mostrada correctamente.

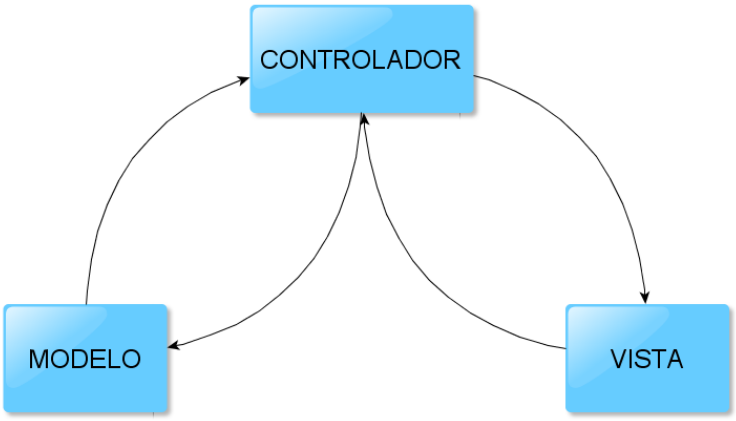


Figura 2: Diagrama del patrón de diseño Modelo vista controlador

2.2 Pasarelas de pago

Una pasarela de pago es, por simplificar un poco el concepto, un canal a través del cual puedes solicitar la realización de una operación bancaria. Su funcionamiento pasa por dos fases. En la primera fase, la pasarela de pago recibe una solicitud de operación. A continuación, la pasarela de pago se pone en contacto con el banco y solicita la ejecución de dicha solicitud. El tercer paso depende del tipo de pasarela que se esté usando. Hay tres tipos de pasarela:

- síncronas: Estas pasarelas de pago no devuelven ningún dato al cliente hasta que el banco ha contestado, ya sea que se ha podido realizar o que no. Mientras no se reciba respuesta, el cliente de dicha pasarela de pagos queda a la espera de respuesta (Ver figura 3 en la página 9).

- asíncronas: Este tipo de pasarelas permite invocar a la pasarela de pago pasándole un parámetro adicional: un callback que será invocado cuando el banco haya terminado la transacción (ya haya sido con éxito o no).(Ver figura 4 en la página 9)

- Híbridas: Este tipo de pasarelas permite ambos modos de operación (síncrono y asíncrono).

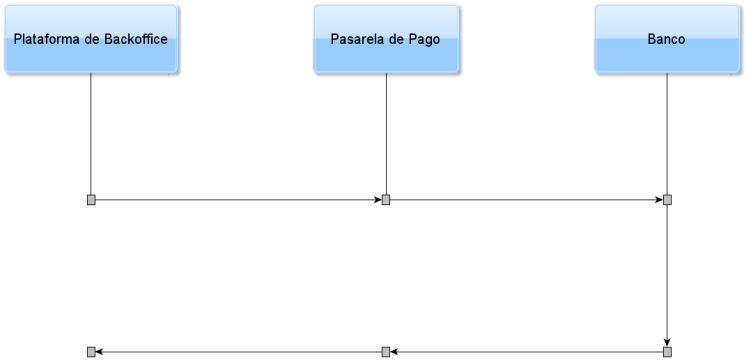


Figura 3: Diagrama de flujo de las pasarelas Síncronas

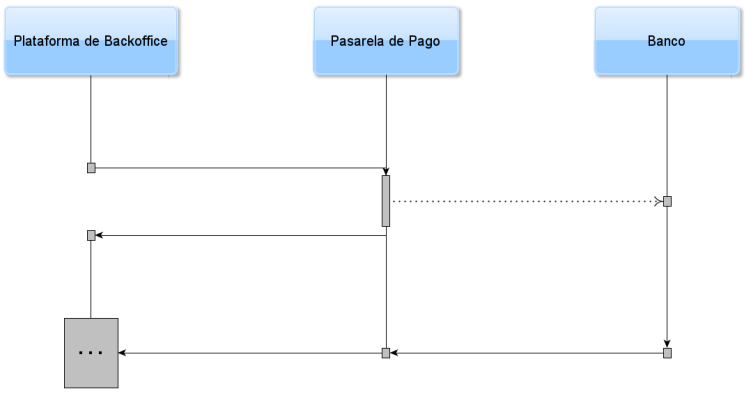


Figura 4: Diagrama de flujo de las pasarelas Asíncronas

La decisión de utilizar WorldPay ha venido dada por parte de telefónica digital por estar presente en su plan tecnológico. Sin embargo, al ir avanzando el proyecto, se propuso un caso de aplicación para el sistema de *backoffice*, la integración con un supuesto servicio de consultoría de salud online que operaría en Españay Brasil. WorldPay no opera en Brasil, por lo que se tuvo que implementar el uso de otra de las pasarelas de pago más conocidas: Adyen. La decisión de usar esta pasarela también vino dada por telefónica digital, pero esta vez no dieron motivos que apoyen la elección de esta pasarela y no otra. Sin embargo, en el uso durante el proyecto, ha demostrado ser una pasarela de pago muy intuitiva de integrar en la aplicación, ofreciendo, por ejemplo, una copia de la pasarela en un sandbox de manera que se pueda comprobar la correcta integración de la aplicación que se está desarrollando con la pasarela sin la necesidad de efectuar operaciones reales que invelucren dinero real. En dicha sandbox también se puede especificar la respuesta que se va a dar ante una petición para controlar que estén correctamente implementados los callbacks tanto de éxito como de error.

2.3 La nube

El concepto de "La nube" es la evolución de ideas surgidas en los años 50, donde Herb Grosch intuyó que en el futuro todo el mundo usaría máquinas vacías conectadas a enormes centros de proceso de datos[3], y los 60, donde John McCarthy postuló que algún día cada suscriptor pagaría sólo por la capacidad que consuma de un sistema muy grande, y que algunos de esos suscriptores podría incluso ofrecer servicios a otros suscriptores[4]. Todas estas visiones de lo que sería el futuro de la informática están hoy día recogidas en la definición de Cloud Computing propuesta por el NIST: "El Cloud Computing es un modelo que permite acceso a través de la red a un conjunto compartido de recursos configurables (como servidores, redes, almacenamiento, aplicaciones y servicios) que pueden ser rápidamente reservados y liberados con un esfuerzo de gestión y una interacción del proveedor de servicios mínimos"[5]

Según el NIST, para que algo pueda ser denominado Cloud computing, debe reunir 5 características fundamentales:

1. Autoservicio bajo demanda: El usuario puede reservar recursos del proveedor unilateralmente, es decir, sin que el proveedor tenga ningún papel en dicha transacción.

2. Amplio acceso a la red: Los recursos están disponibles en la red y son accesibles mediante mecanismos estándar.

3. Reserva de recursos: Los recursos del proveedor están creados y son servidos siguiendo un modelo multi-tenant. Esto quiere decir que los recursos pueden ser accedidos por varios usuarios a la vez (cada uno a sus propios recursos). Además dichos recursos han de poder ser asignados y reasignados dinámicamente según las necesidades de los usuarios.

4. Elasticidad: El proceso recién descrito (asignación dinámica de recursos) ha de ser llevado a cabo rápidamente de manera que sea un proceso totalmente transparente al usuario, que debería acabar con la sensación de que los recursos de que dispone son ilimitados.

5. Medida del servicio: Para poder garantizar los puntos anteriores, el sistema cloud debe poseer alguna forma de medir si los recursos de que dispone un usuario son suficientes para la actividad que estzá llevando a cabo o si, por el contrario, hay que proveerle de más recursos.

También cabe destacar que dentro de la denominación "Cloud computing" hay varios tipos que conviene distinguir, atendiendo a dos tipos de clasificaciones: el tipo de servicio de que proveen al usuario y quienes pueden acceder a los recursos. Según la primera clasificación, se pueden encontrar los siguientes tipos:

- Software as a Service (SaaS): Este modelo de cloud consiste en que el proveedor instala cierto software en un servidor y el cliente puede consumir ese software. Algunos de los ejemplos que se pueden dar de este tipo de cloud son Netflix (un servicio de visionado de videos por streaming), Gmail (un servicio de correo electrónico), Dropbox (Un servicio de almacenamiento y sincronización de datos en la nube) o Google docs (Un servicio de edición de documentos de manera colaborativa en la que los mismos documentos están de por sí almacenados en la nube).

- Platform as a Service (PaaS): Este modelo de cloud provee de mayor libertad que el anterior. En este caso, el proveedor también instala una capa de software, pero en este caso para facilitar el desarrollo de aplicaciones propias. Por tanto, estos software son bases de datos, servidores web, etc. En este tipo de Cloud tendríamos encuadrado, por ejemplo, Heroku PaaS o Google App Engine.

- Infrastructure as a Service (IaaS): Este modelo de cloud es el que provee de mayor libertad al usuario. Consiste en alquilar a los usuarios máquinas virtuales dentro de los servidores del proveedor. De cara al usuario, es como tener un servidor dedicado al que, además se le puede añadir o quitar tanto

capacidad de cómputo, de disco duro y Ram dinámicamente según sea necesario en cada instante. El mayor ejemplo de este tipo de cloud hoy día es el "Elastic Cloud" de Amazon (EC2), aunque existen otros jemlos como pueden ser DynDNS o Google compute Engine.

Según la segunda clasificación podemos encontrar otros dos tipos de Cloud: públicos y privados:

- Públicos: Los sistemas cloud P'ublicos son aquellos que están desplegados de manera que cualquier usuario pueda acceder y consumir sus recursos. El proveedor del servicio puede cobrar por su uso o no.
- Privados: Los sistemas cloud privados son aquellos que se despliegan dentro de una organización con el fin de que sean consumidos por el propio personal de la organización. Para evitar el acceso por parte de gente externa, se suele colocar detrás de un firewall, que denegaría el acceso

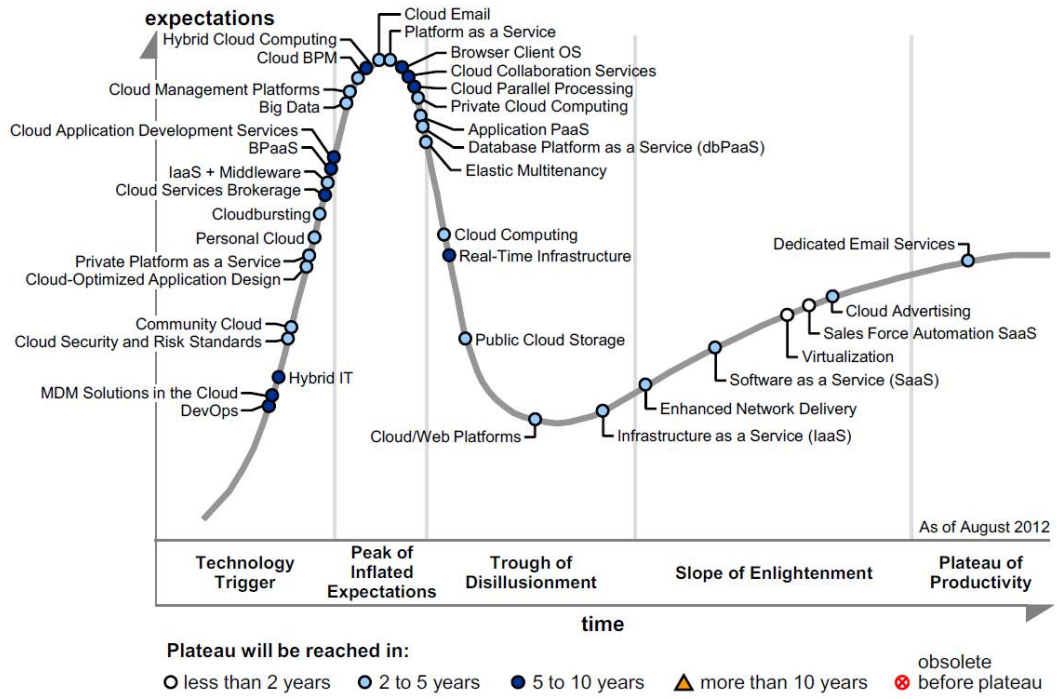
2.3.1 Diagrama del ciclo de Gartner

Cabe destacar la importancia que se prevee van a ganar en los próximos años los PaaS orientados al desarrollo de Software. Para ello, me voy a apoyar en el diagrama de Gartner para el hype de tecnologías Cloud en 2012 (ver figura 5).

Este diagrama está elaborado por una empresa especializada en prospección de información sobre el estado de desarrollo e importancia de las tecnologías más punteras del momento. El gráfico se divide en cinco partes [6]:

- Lanzamiento: Representa la etapa en la que una tecnología es presentada y los empleados encargados de la prospección tecnológica en las empresas empiezan a mostrar su interés por la evolución de dicha tecnología.
- Pico de expectativas sobredimensionadas: En esta fase se produce un entusiasmo excesivo por el uso de una tecnología, que normalmente viene acompañado de un gran número de proyectos de innovación que si son poco realistas y/o demasiado ambiciosos suelen fracasar.
- Abismo de desilusión: Las tecnologías entran en esta etapa cuando no cumplen las espectativas que se generaron en la etapa anterior. La tecnología deja de "estar de moda" y deja de ser usada en proyectos.
- Rampa de consolidación: En esta etapa las empresas vuelven a retomar el uso de una tecnología a fin de comprobar hasta qué punto se ha desarrollado y qué beneficios proporciona su uso, desde una perspectiva mucho más realista que en la etapa de expectativas sobredimensionadas.

Figure 1. Hype Cycle for Cloud Computing, 2012



Source: Gartner (August 2012)

Figura 5: Hype de tecnologías Cloud para 2012

- Meseta de Productividad: En esta última etapa, la tecnología ha demostrado sus beneficios y ha sido incluida en los planes tecnológicos de las empresas del sector. La tecnología se vuelve más estable.

Es importante mencionar que no todas las tecnologías cumplen el ciclo completo. Es posible abandonar el diagrama en cualquiera de sus fases si una tecnología, por el motivo que sea, deja de ser usada y no se crean expectativas nuevas en torno a ella. Cabe decir que también es posible volver a entrar en el ciclo después de haberlo abandonado.

Como podemos ver en dicho diagrama, el item llamado "Cloud Application Development Services" en 2012 estaba a punto de abandonar la zona de lanzamiento tecnológico, por lo que a día de hoy intuimos que dicho item debe estar en la zona de expectativas sobredimensionadas (No se puede comprobar, ya que aún no ha sido liberado el diagrama para el año 2013.

En vista a ésto, la solución elegida para este trabajo debería ser la ofrecida por un PaaS, ya que el desarrollador puede abstraerse de detalles de la administración del servidor y a su vez participar de las facilidades que ofrece al tener ya pre instalados todos los software externos necesarios, lo que proporciona la mayor ventaja que podemos extraer de la tecnología cloud, evitando tener que dedicar recursos a ocuparnos de instalar y gestionar todo el "Software satélite" de nuestra aplicación (lo que tendríamos que hacer de tratarse de un IaaS). En el caso de este tfg no tendría sentido el uso de un cloud SaaS, ya que el objetivo no es ser clientes de una serie de aplicaciones ya establecidas, sino desarrollar una aplicacion y servirla en cloud. En esta prueba de concepto se ha consensuado probar Heroku PaaS, que tiene la peculiaridad de ser un cloud que extrae sus recursos de otro cloud (los toma de EC2). El motivo de esta decisión es la existencia de unos componentes software creados en Heroku, los llamados Heroku addons, que extienden aun más la capacidad de ofrecer una capa de software al desarrollador.

A continuación haremos un análisis de dichos componentes software.

2.4 Heroku addons

Heroku addons es una de las cosas que diferencia el PaaS de Heroku del resto; consiste en una serie de extensiones que puedes integrar en tu aplicación para dotarla de nuevas funcionalidades sin tener que programarlas ni configurarlas. Estas extensiones incluyen las siguientes categorías:

- Almacenamiento de datos: Es una colección de 21 extensiones para dotar de base de datos a tu aplicación, así como de una politica de backups.
- Móvil: Es una colección de 5 extensiones que permiten adaptar tu contenido multimedia a formatos válidos para móviles, así como enviar notificaciones a los mismos.
- Búsqueda: Colección de 9 extensiones que permiten la integración de un motor de búsqueda e indexaciónen tu aplicación.
- Registro: Es una colección de 4 extensiones que permiten guardar un registro de lo que está pasando en tu aplicación. Son de especial utilidad cuando ocurre algún fallo en la aplicación y se quiere depurar.
- Email y sms: Es una colección de 7 extensiones que permiten el envío de emails y sms de manera fiable desde tu aplicación.
- Workers y queues: Es una colección de 8 extensiones que permiten que la aplicación sea más escalable mediante el uso de sistemas de colas que almacenan las peticiones hasta que éstas pueden ser resueltas.
- Analisis: Es una colección de 10 extensiones que permiten añadir un sistema de métricas para obtener información sobre el rendimiento o las estadísticas de uso de la aplicación.
- Cache: Es una colección de 9 extensiones que dotan a la aplicación de un sistema de caché.
- Monitorización: Es una colección de 15 extensiones que permiten monitorizar el sistema en busca de errores. Estos errores serán convenientemente almacenados y el administrador de la aplicación será informado de ella.
- Medios: Es una colección de 10 extensiones que permiten el tratamiento de archivos multimedia desde tu aplicación. Dentro de esta categoría encontramos extensiones que permiten cosas tan diversas como almacenar imágenes, crear ficheros pdf, crear ficheros de excel o alacenamiento y reproducción de vídeo.
- Pagos: En esta categoría sólo existe una extensión, Spreedly, que permite realizar cobros sin tener que implementar las llamadas a las pasarelas de pago.
- Utilidades: En esta categoría encontramos una colección de 16 extensiones que no encajan con las categorías anteriores. Con ellos se pueden hacer cosas como creación de subdominios DNS, Enrutar el trafíco generado desde tu

aplicación por una IP estática, buscar e instalar actualizaciones de librerías y dependencias que uses o añadir soporte para conexiones ssl.

La mayor parte de estas extensiones están desarrolladas por la comunidad que hay detrás de Heroku y están disponibles para ser usadas por todo el mundo. Los que no están disponibles para todo el mundo es porque están aún en fase beta y para poder usarlos necesitas haber sido aceptado como parte de la comunidad de *betatesting* de Heroku. Esta característica de Heroku es la ms atractiva y prueba de ello es la cantidad de extensiones que, como hemos visto, existen ya y la cantidad de extensiones que siguen en desarrollo.

3 Desarrollo

El desarrollo de este TFG ha estado dividido en varias partes, siguiéndose en todas una metodología común de reducción de la incertidumbre existente. Hay que recordar en todo momento que la solución desarrollada es una prueba de concepto en la que los requisitos cambiaban con frecuencia, por lo que se ha tenido especial cuidado en el diseño del código para que un cambio en los requisitos afectara lo menos posible al trabajo ya realizado. De ahí la necesidad de reducir, como decía, la incertidumbre existente.El código de la plataforma de backoffice puede obtenerse en su repositorio de github (`https://github.com/PDI-DGS-Protolab/backoffice_process_manager`). En el resto del capítulo se va a profundizar en qué se llevó a cabo en cada parte de la aplicación.

3.1 Esquema de la arquitectura

En este apartado se va a explicar la arquitectura final de la aplicación, que se puede observar en la figura 6

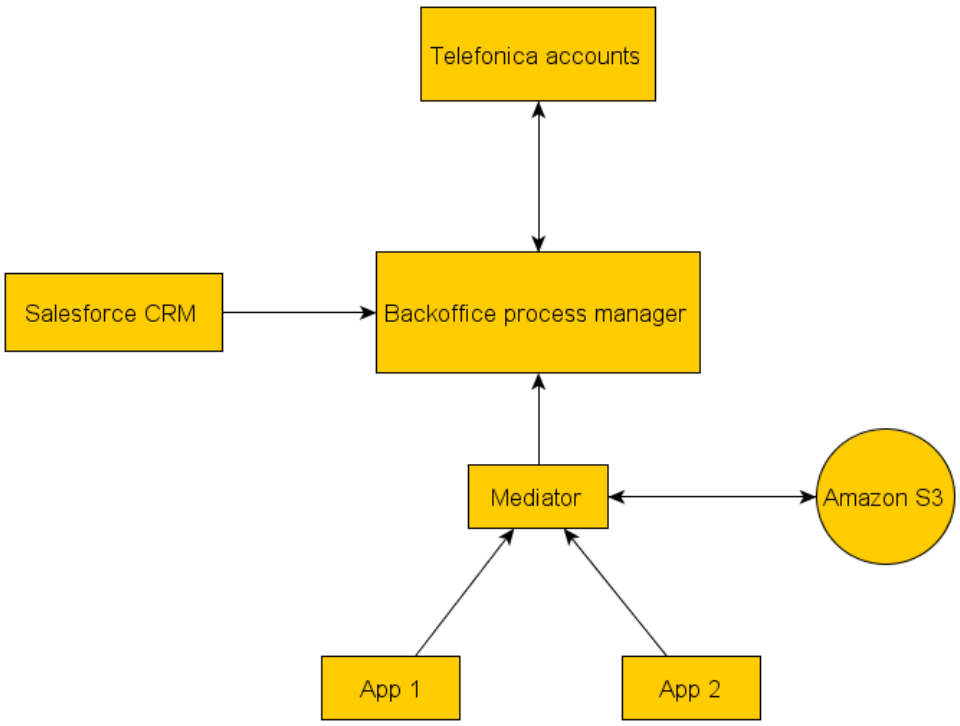


Figura 6: Diagrama de la arquitectura implementada

Las distintas partes que forman el sistema de la imagen son:

- Salesforce CRM: Salesforce es uno de los CRM más utilizados hoy en día. En cuanto a su uso en el sistema objetivo de este tfg, se centra en dos funciones: la adquisición de usuarios y la consulta del estado del pago de los usuarios por parte del personal de contabilidad.

- Telefónica accounts: Es el servicio de login y credenciales de Telefónica. Toda persona que haya contratado alguna vez un servicio de telefónica se encuentra en la base de datos de este servicio. Nuestra aplicación simplemente consumirá la API proporcionada por Telefonica accounts tanto para introducir usuarios que previamente no existían como para recuperar datos sobre un/os usuario/s. Durante el desarrollo se ha contado con una réplica exacta del servicio en un sandbox, de manera que se ha podido realizar las pruebas de integración pertinentes sin que el servicio real corriera ningún riesgo.

- App 1, App 2, etc: Son los servicios que pueda querer implementar telefónica en un futuro. Como se puede apreciar en el diagrama, se ha operado un gran cambio con respecto a la arquitectura antigua: Se ha sustituído un modelo de polling (En el que el servidor pide mediante las distintas apis los eventos facturables a los servicios) por un modelo de push (en el que el módulo *Mediator* está a la escucha de los eventos facturables que le vayan llegando).

- *Mediator*: Este es uno de los dos grandes módulos que se han programado. Su función consiste en recibir los eventos facturables, almacenarlos en Amazon S3, y cada periodo de facturacion calcular cuanto hay que cargar a cada usuario. Finalmente pasaró ese valor al *backoffice process manager* (Se explicará en la sección 3.4 , página 27).

- Amazon S3: Es el servicio de almacenamiento en la nube de Amazon. Forma parte de los llamados Amazon web services (AWS). En este caso se usará para almacenar los sdr que serán procesados durante el siguiente periodo de facturación

- *backoffice process manager*: Éste es el otro módulo que ha sido implementado como parte de este TFG. Su papel consiste en cobrar a cada usuario por los servicios que haya utilizado ese periodo facturable, asegurando la robustez del servicio (Se explicará a fondo a continuación).

3.2 Backoffice process manager

Este es el elemento clave de este TFG: La creación de la plataforma de *backoffice*. Consiste en un proyecto de Django que, tal y como vimos en el diagrama de la

arquitectura, interactúa con todas las piezas clave del sistema: El CRM de salesforce, el Mediador, el servicio de Telefonica accounts y las pasarelas de pago.

En cuanto a su funcionalidad, se puede resumir en los tres casos de uso que expongo a continuación (junto con su diagrama BPM) y que he estado encargado de implementar:

En el primer caso de uso (figura 3.2, página 22) se puede ver cómo un usuario crea una cuenta en el sistema (que terminará desembocando en telefónica accounts). En este caso de uso el *backoffice process manager* se limita a crear la cuenta de usuario en telefonica accounts,si no tenía ya una, y pedir a dicho servicio el id de usuario que se le ha asignado para devolverlo al CRM a fin de que éste almacene la información de ese usuario en su base de datos. Cuando acaba este proceso, se redirige al usuario a la página donde da comienzo el siguiente caso de uso.

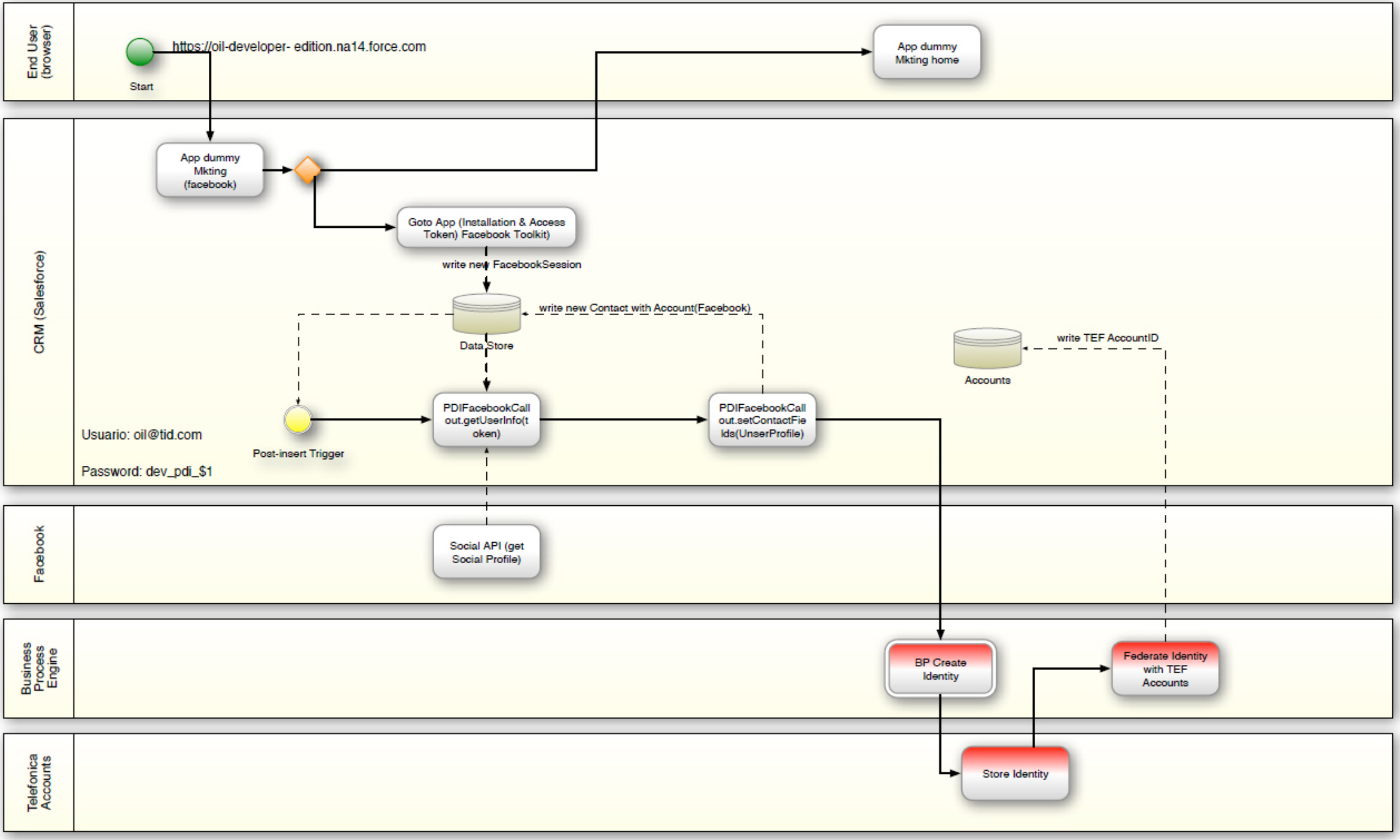
En el segundo caso de uso (figura 3.2, página 23), se muestra el caso correspondiente a la realización de un microcobro a la cuenta bancaria que ha introducido el usuario para verificar que es válida. Para ello, lo primero es tomar los datos bancarios del usuario y que éste acepte los términos y condiciones de uso. A continuación se solicita a la pasarela de pago que realice el microcobro de verificación y, en caso de tener éxito, se requerirá a dicha pasarela el código de pago recurrente, código que será necesario más adelante para poder realizar los pagos recurrentes. Una vez obtenido, se almacenará a fin de poder realizar el resto de operaciones (en nuestro caso, las de cobro).

Finalmente, en el último caso de uso (figura 3.2, página 24) se puede ver la modelización en tareas del caso de uso de realizar los pagos recurrentes. Es un caso de uso que sigue una secuencia lineal de tareas.

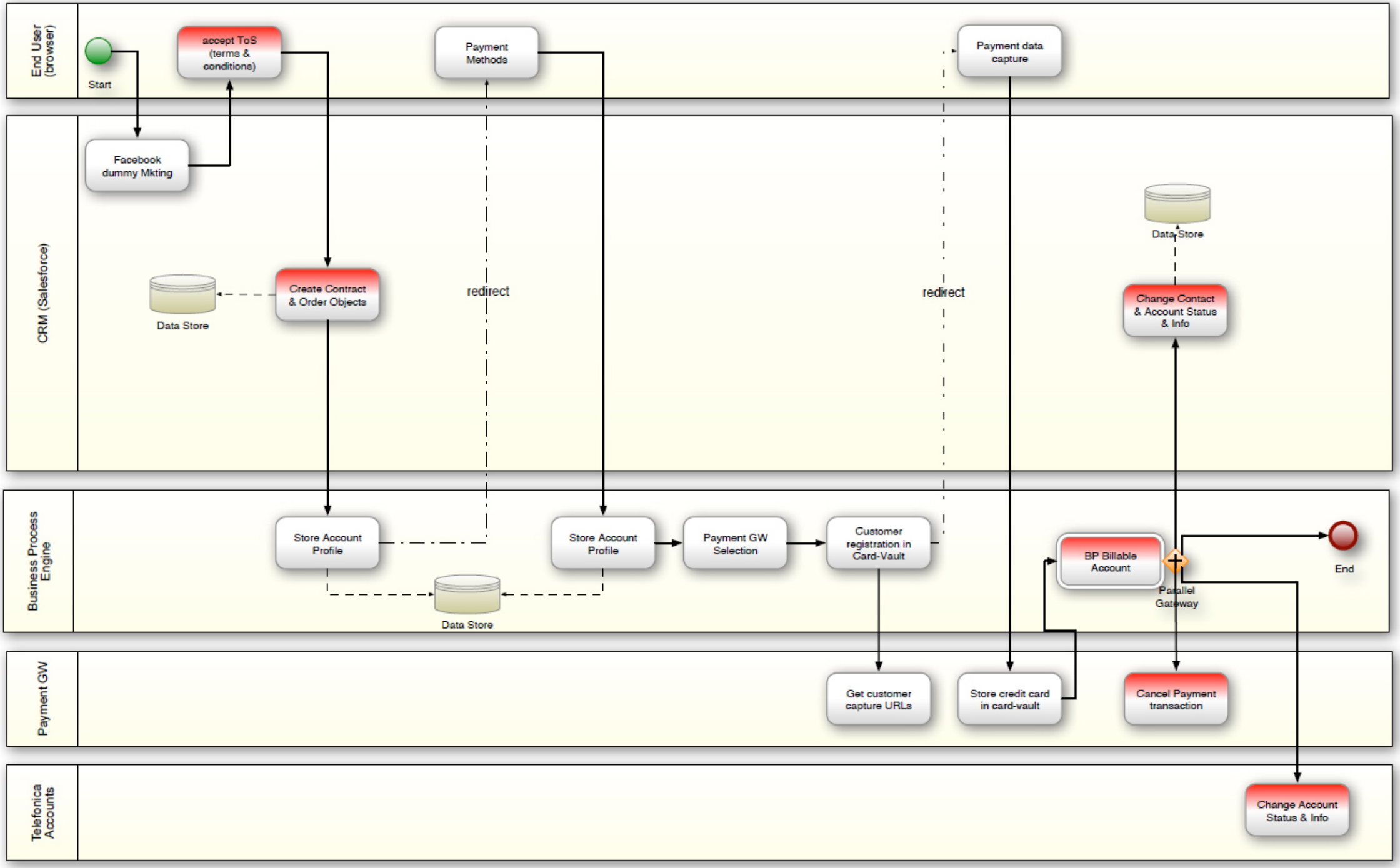
La primera tarea (llamada "rating") consiste en la aplicación de las tasas correspondientes al país donde se han consumido los bienes digitales que vamos a cobrar. A continuación se pasa a la tarea "invoice", que es el envío de un correo en el que se adjunta la factura al usuario. Después hay que realizar el cobro dándole a la pasarela de pagos el identificador de la aplicación, la cantidad a cargar, el código de pagos recurrentes que nos devolviera cuando verificamos la cuenta bancaria y la/s url/s de callback (Adyen solo usa una, mientras que WorldPay necesita que se le envíen 2).

Finalmente, cuando la pasarela de pago haya realizado el cobro, invocará la url correspondiente al estado de terminación (Si se trata de Adyen, invocará siempre la misma url pasándole parámetros distintos para cada estado de terminación).

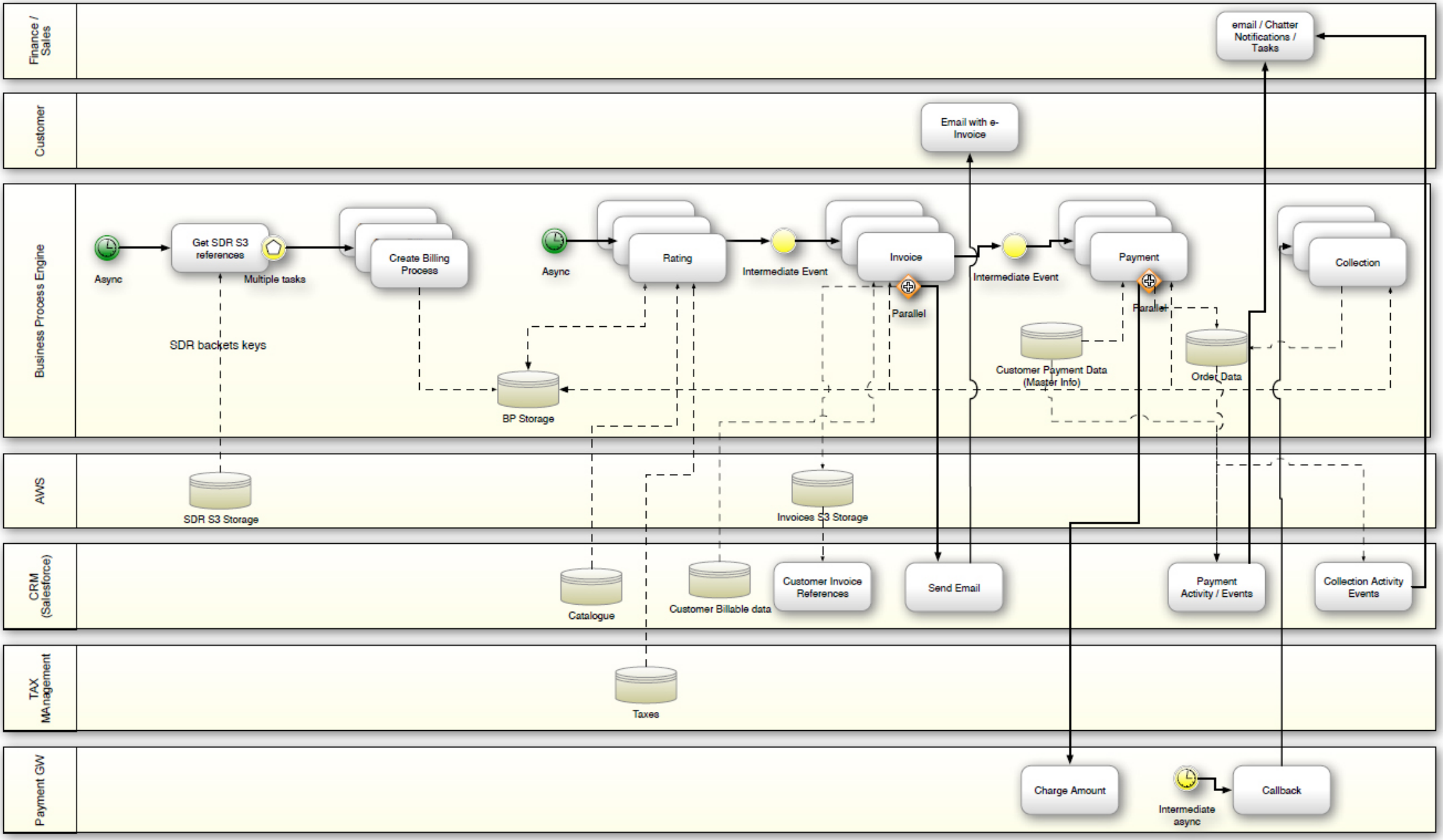
BPMN1 - CUSTOMER ACQUISITION



BPMN2 - ACQUIRE PAYABLE ACCOUNT (RECURRENT)



BPMN3 - POST PAID - RECURRENT



Ahora bien, si se atiende al modelo de procesos que acabamos de describir, si el servidor perdiera, por ejemplo, la conexión a internet justo tras mandar la orden de pago a la pasarela, en el servidor esa tarea constaría como "No completada", con lo que habría que decidir si volver a lanzar el proceso (asumiendo el riesgo de cobrar dos veces) o no volverlo a lanzar (corriendo el riesgo de no cobrar). Para evitar esta situación, se ha diseñado el modelo de datos que se verá' en la proxima sección, y el protocolo de actuación que se definirá en la siguiente.

3.2.1 Modelo de datos

De acuerdo con el propósito que tiene el *backoffice process manager*, se hace sumamente necesario que la base de datos sea robusta y que se consoliden en ella los datos después de que cada parte de una operación termine. Por ello se me encargó que diseñara un modelo de datos que cumpliera con dicha especificación. Como resultado, se ha terminado eligiendo un modelo de datos como el que se puede ver en la figura 7

En él se puede apreciar una estructuca jerárquica en la que todo proceso se subdivide en subprocesos y éstos se dividen a su vez en tareas. Las tareas son la unidad mas pequeña de trabajo en nuestro sistema y se corresponden con las burbujas pequeñas de los diagramas BPM. Vistos en *backoffice process manager*. A continuación paso a explicar cada campo de dicho modelo:

- Process
  - tef_account: el identificador de telefonica accounts del usuario dentro a quien se está' cobrando con este proceso
  - name: Nombre del proceso ("tipo" de proceso)
  - start: Timestamp del momento en que se inició el proceso.
  - end: Timestamp del momento en que terminó el proceso. Si aún no ha terminado, estará vacío.
  - initial_data: Datos con los que se ha ejecutado el proceso.
  - status: estado de terminación del proceso. Puede ser uno de los siguientes: OK, ERROR, CANCELED, PENDING
- SubProcess
  - Process: Identificador del proceso al que pertenece.
  - name:Nombre del subproceso ("tipo" de subproceso)
  - start: Timestamp del momento en que se inició el subproceso.

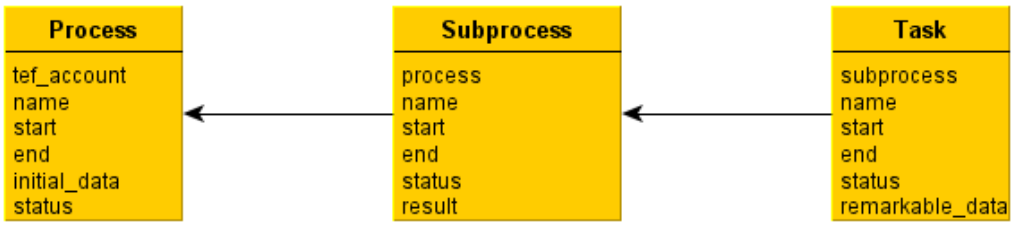


Figura 7: Diagrama del modelo de datos

  - end: Timestamp del momento en que terminó el subproceso. Si aún no ha terminado, estará vacío.
  - status: estado de terminación del proceso. Puede ser uno de los siguientes: OK, ERROR, CANCELED, PENDING
  - result: Datos adicionales sobre el estado del subproceso

- Task
  - subprocess: Identificador del subproceso al que pertenece.
  - name:Nombre de la tarea ("tipo" de tarea)
  - start: Timestamp del momento en que se inició la tarea.
  - end: Timestamp del momento en que terminó la tarea. Si aún no ha terminado, estará vacío.
  - status: estado de terminación del proceso. Puede ser uno de los siguientes: OK, ERROR, CANCELED, PENDING
  - remarkable_data: datos adicionales sobre el estado del proceso.

3.2.2 Modelo de ejecución

Para garantizar que el modelo de datos propuesto se rellena con los datos adecuados y que no se queda en ningún momento inconsistente con la realidad, se ha utilizado una aplicación llamada Celeryd (en su versión para Django, llamada django-celeryd). Esta aplicación se autodenomina "Cola de tareas", ya que provee de un sistema asíncrono de ejecución de tareas en las que la salida proporcionada por una de las tareas es conceptualmente la entrada de la siguiente.

django-celery provee de una API muy sencilla a través de la cual se especifica qué funciones hay que invocar y en qué orden para completar un proceso más complejo. Simplemente hay que añadir a las tareas el decorador @task y en el método en el

que se lanza el proceso crear una secuencia ordenada formada por los nombres de las funciones que se van a invocar separados por el caracter "|". Tras esto, se ejecutará una función propia de django-delery que se encarga de ir lanzando dichas tareas.

Sin embargo, lo explicado hasta ahora no justifica la consistencia de la base de datos. Dicha justifiación consiste en el hecho de que dentro del código de cada tarea incluimos al final un bloque cuyo objetivo es consolidar sus resultados en la base de datos y marcar su propio estado como 'OK'.

De esta manera, y permitiendo que se pueda acceder a estos datos desde Salesforce se facilita la posibilidad de relanzar tanto manualmente (en el caso de que haya que comprobar si una tarea errónea se llegó a ejecutar siendo ésta crítica, como el envío de la orden de pago a la pasarela o que sea algo que tiene que valorar alguien del departamento de ventas, como la no existencia de fondos en la cuenta a la que hay que cobrar) como automáticamente (en el caso de que se haya producido un error en una fase no crítica) las tareas que hayan fallado por el motivo que sea.

3.3 Definición de un nuevo SDR

Uno de los problemas que tenía el sistema que seguía telefónica a la hora de mantener servicios eran los SDR que utilizaban. Como se vió en la introducción, el formato de los mismos no era el más apropiado para el uso que se les estaba dando. Por tanto, hay que definir un nuevo modelo de SDR que mejore el rendimiendo de la aplicación.

La primera elección que se tuvo que realizar fue si seguir usando xml como formato de los SDR o pasar a usar JSON. La dificultad de los formatos xml es que analizar y extraer los datos de un fichero xml es bastante complejo y requiere del uso de bibliotecas externas (o en último caso, implementar una biblioteca que sea capaz de analizar xml). Sin embargo, python lleva en su núcleo una biblioteca denominada json que permite convertir de manera muy rápida un json a un diccionario, una de las estructuras de datos básica en Python.

En cuanto al espacio que ocupan, la diferencia es casi inapreciable y, en todo caso, redunda en favor del uso de json. Estos son los motivos principales por los que se decidió migrar el formato a json. Una vez hecho esto, decidimos crear dos tipos de SDR;

Por un lado, los SDR que generan los servicios que telefónica tenga desplegados (uno por evento), cuyo formato será el siguiente:

- Event
  - eventId: El identificador de evento dentro del servicio.
  - accountId: El identificador en telefonica accounts del usuario a quien pertenece este evento

  - serviceId: El identificador del servicio que ha generado este evento
  - timestamp: Cuándo ha ocurrido este evento
  - description: Descripción complementaria que iró incluida en la factura.
  - extraData: Información extra que se quiera hacer constar.
  - rate
    * ud: Unidades de consumo (dependiendo del servicio las unidades serán unas u otras: días, MB, usos, etc)
    * pricePerUnit: Precio por unidad de consumo

Con ese formato se guardan los eventos sin procesar en Amazon S3. Sin embargo, cuando se lanza el proceso de facturación, al *backoffice process manager* hay que pasarle un unico valor a cobrar. Por tanto se hace necesario otro formato que integre todos los eventos de un periodo facturable y un módulo que realice la integración de los SDR del primer tipo en los del segundo. El módulo que se encargará de dicha tarea será el *Mediator* (al que se hará referencia en la siguiente sección). Por otro lado, el formato nuevo agregado será el siguiente:

- ListEvents
  - serviceId: El identificador del servicio que ha generado estos eventos
  - accountId: El identificador en telefonica accounts del usuario a quien pertenece este evento
  - timeStamp: el timestamp del periodo facturable
  - events : List<Event> : lista de eventos del periodo

3.4 Mediator

En esta sección se va a cubrir todo lo relacionado con el último objetivo que se añadió al proyecto: el desarrollo de la plataforma de mediación que va a proporcionar los datos al servicio de *backoffice* que ha sido desarrollado cuando se lance un proceso de cobro.

La labor del mediador es, como ya se mencionó en la introducción de este capítulo, recoger los datos de uso de los distintos servicios de los que provee Telefónica y unir dichos datos de uso en un fichero único por cliente. A pesar de la sencillez con que parezca que se puede acometer esta tarea, no es algo trivial, ya que hay que tener presente que el escenario que se pretende conseguir es que, por poner un ejemplo, cada llamada que haga una persona sea registrada inicialmente en un fichero distinto y éstos serán los que se envíen para mediar. Y ocurrirá lo mismo para cada

uno de los servicios existentes y por existir. Esto desemboca en una situación en la que nuestro sistema deberá soportar una gran carga. Por tanto, se debe disear primero una arquitectura que permita recibir y tratar la gran cantidad de peticiones que se van a producir.

Telefónica incluyó el requisito de utilizar una técnica conocida como "Time boxing" en la que la mediación se va a realizar en varias etapas temporales hasta conseguir los ficheros que se buscaban (separados por usuario).

En la primera etapa, el mediador almacena los sdr que le llegan organizados en directorios según el timestamp de su llegada al mediador con el siguiente convenio de nombres (que deben respetar los servicios emisores del sdr) < identificador del usuario >. A su vez, los directorios usarán el siguiente convenio de nombres:

id_servicio.AAAA.MM.DD.hh

Donde:

- id_servicio es el identificador único del servicio al que pertenece el sdr
- AAAA es el año
- MM es el mes
- DD es el día
- hh es la hora

Cada hora se lanzará un proceso con el que se hará la mediación de todos los sdr generados para el mismo usuario durante la hora anterior, almacenando el resultado en otro árbol de directorios, esta vez ordenado por usuario.

El segundo paso consiste en mediar una segunda vez, pera ahora todos los ficheros de un día para tener en un solo fichero todos los eventos de un cliente en cada día. Finalmente, una vez al mes se lanzará el proceso de cobro a los usuarios, durante el cual se realizará la última mediación, en la que se juntarán todos los sdr del mes por usuario (y el resultado será el SDR que se enviará para cobrar a la plataforma de *backoffice*).

Aunque parezca que esta solución no reduce la carga del sistema, sino que incluso la hace mayor (y globalmente es cierta dicha afirmación), lo que se consigue con la técnica de time boxing es repartir la carga que se generaría tradicionalmente en el momento de lanzarse el proceso de cobro a lo largo de todo el mes, mediando parcialmente todo lo que se puede para que la operación de cobro pueda realizarse de la manera más eficiente (y sobre todo escalable) posible.

3.5 Estudio de los Addons de Heroku

El último de lo objetivos de este TFG que falta por cubrir es el estudio sobre el ahorro de *time to market* que se puede conseguir al desarrollar aplicaciones utilizando las piezas de software que ofrece Heroku bajo el nombre de *addons*.

En el uso que se les ha dado durante la implementación de los primeros objetivos de este TFG hemos utilizado dos de ellos: el que ofrece soporte para el uso de PostgreSQL y el que ofrece soporte para el uso de RabbitMQ como *broker* para el sistema de colas de tareas. El último se encuentra aún en fase beta, y para su uso es necesario solicitar credenciales de *tester*.

La realidad es que gran parte de dichos addons están aún en fase de pruebas, por lo que, si tuvieéramos que colocar el uso de éstas piezas de sofware en el ciclo de Gartner, con toda probabilidad se encontrarían en la primera etapa, la de lanzamiento.

Por otro lado, conviene destacar un punto negativo del uso de los *addons* de Heroku: los precios. La mayor parte de los *addons* provee de una versión gratuita (muy limitada en cuanto a prestaciones) y varias versiones de pago. El problema radica en el precio desmesurado de dichas versiones de pago según se van incluyendo características. Para ilustrar esta variación de precios, se van a exponer los de tres de los *addons*:

- Postgres:
  - gratis: esta versión está limitada a 10.000 filas en la base de datos y a soportar 20 conexiones. No incluye la realización automática de backups ni el uso de RAM para agilizar las búsquedas.
  - $9/month: esta versión está limitada a 10.000.000 de filas en la base de datos y a soportar 20 conexiones. No incluye la realización automática de backups ni el uso de RAM para agilizar las búsquedas.
  - $50/month: Número ilimitado de filas, limitada a 500 conexiones y uso de RAM limitado a 410 MB. Incluye la realización automática de backups.'
  - $100/month: Lo mismo que la anterior, pero con un uso de RAM limitado a 819 MB
  - $200/month: Lo mismo que la anterior, pero con un uso de RAM limitado a 1.7 GB
  - $400/month: Lo mismo que la anterior, pero con un uso de RAM limitado a 3.75 GB
  - $800/month: Lo mismo que la anterior, pero con un uso de RAM limitado a 7.5 GB

  - $1600/month: Lo mismo que la anterior, pero con un uso de RAM limitado a 15 GB
  - $3200/month: Lo mismo que la anterior, pero con un uso de RAM limitado a 34 GB
  - $6400/month: Lo mismo que la anterior, pero con un uso de RAM limitado a 68 GB

- CloudAMQP (Rabbit MQ):
  - gratis: Máximo de 30 MB de mensajes al mes. No incluye soporte para conexiones por ssl, STOMP ni MQTT. Limitado a 3 conexiones concurrentes.
  - $19/month: Igual que la anterior, pero con 2 GB de mensajes al mes y limitado a 6 conexiones concurrentes.
  - $99/month: Igual que la anterior, pero con 20 GB de mensajes al mes y limitado a 20 conexiones concurrentes. Incluye soporte para ssl.
  - $299/month: Igual que la anterior, pero con 100 GB de mensajes al mes y limitado a 100 conexiones concurrentes.
  - $499/month: Sin límite de mensajes ni de conexiones concurrentes. Incluye los protocolos STOMP y MQTT.

- Redis:
  - gratis: Incluye un tamaño de memoria de 20 MB, un límite de una base de datos y un límite de 10 conexiones.
  - $10/month: En esta versión se incluye un tamaño de 100 MB, número ilimitado de bases de datos, límite de 256 conexiones, *backups* diarios en Amazon S3, sistema de replicación y de *failover*.
  - $25/month: Contiene lo mismo que la versión anterior, pero limitando la memoria a 200 MB.
  - $52/month: Contiene lo mismo que la versión anterior, pero limitando la memoria a 500 MB y las conexiones a 512.
  - $102/month: Contiene lo mismo que la versión anterior, pero limitando la memoria a 1 GB y las conexiones a 1024.
  - $252/month: Contiene lo mismo que la versión anterior, pero limitando la memoria a 2.5 GB y conexiones ilimitadas.
  - $499/month: Contiene lo mismo que la versión anterior, pero limitando la memoria a 5 GB

- $899/month: Contiene lo mismo que la versión anterior, pero limitando la memoria a 10 GB
- $3500/month: Contiene lo mismo que la versión anterior, pero limitando la memoria a 50 GB. Permite el uso de varios núcleos para mejorar la escalabilidad.

Como se ha podido comprobar, las versiones gratuitas de los *addons* sirven para el desarrollo de la aplicación, pero si se quiere pasar ésta a producción, quizá sea más conveniente buscar otras alternativas que no requieran tales desembolsos de dinero para conseguir unas prestaciones adecuadas.

4 Conclusiones

Para terminar, se exponen los resultados obtenidos con trabajo realizado y las conclusiones que se pueden extraer a partir de los mismos.

4.1 Resultados

Lo primero que hay que mencionar es que la prueba de concepto de la solucin de *backoffice* est desplegada en Heroku y accesible públicamente, tanto la parte de captación de clientes mediante facebook (`http://goo.gl/a67RP`) como la aplicacion que permite iniciar el proceso de facturación (`http://backoffice-process-manager.herokuapp.com/`).

Por otro lado, después de haber realizado demostraciones del funcionamiento de la plataforma y de lo simple que sería ofrecer servicios nuevos si se adopta su uso, los superiores de los jefes del proyecto han quedado bastante impresionados y están evaluando la posibilidad de introducir realmente una versión más compleja de esta plataforma en su estructura.

Finalmente se ha comprobado cmo el uso de los heroku addons reduce ampliamente el time-to-market del servicio, evitándonos tener que instalar, gestionar y configurar todas las tecnologías externas que necesitemos usar (bases de datos, sistemas de logging o monitorización).

4.2 Conclusiones

Se analizan a continuación las conclusiones que se han extraído durante la realización de este TFG. Primero se hablará' sobre la experiencia de desarrollar sobre plataformas PaaS y su estado actual. A continuación, y aunque parezca fuera del alcance de este trabajo, se hablará sobre la suma importancia de seguir una buena metodología en el manejo de git. A continuación se hablará sobre la experiencia con Django y finalmente se hará una valoración sobre los resultados del trabajo.

En el desarrollo de este proyecto se ha puesto de manifiesto la facilidad de trabajo que se consigue al usar una de estas plataformas (Heroku). Gracias a ella no se ha tenido que tener en cuenta nada más que el código que se estába produciendo, evitando tener que compilar, instalar, configurar y desplegar herramientas como el Broker de rabbitMQ, postgreSQL, celeryd o gunicorn.

Pero, como contrapartida, el desarrollo sobre un PaaS requiere que se pruebe el código desplegándolo en la misma plataforma, ya que, de otra manera, se elimina la ventaja que supone poder evitar instalar todas esas tecnologías, pues para probarlas en local habría que instalarlas obligatoriamente.

El proceso de despliegue de la aplicacion en Heroku es rápida (aunque se producen unas variaciones muy grandes en el tiempo de respuesta de Heroku de un despliegue a otro, probablemente debido a la latencia de la red en los distintos momentos en que se ha desplegado). Ésto en un principio no parece una desventaja grave, aunque empieza a serlo cuando se junta con lo anterier. Para probar cada funcionalidad, y cada vez que se resuelva un error hay que hacer un despliegue, lo que al final termina consumiendo bastante tiempo.

Otra de las conclusiones que se han extraído de la realización de este proyecto ha sido la necesidad de usar una meodología común a todo el equipo en el uso de herramienta de control de versiones. Aunque a simple vista este punto no tenga que ver con el poyecto, si lo tiene, ya que uno de los objetivos del mismo era medir ahorro de time-to-market y una buena metodología en el uso de git puede permitir el ahorro de mucho tiempo de desarrollo, pues se evitan la mayor parte de los conflictos de ficheros que por algun motivo han sido modificados por varios desarrolladores a la vez. Esta metodología incluye la política de creación y mantenimiento de ramas de desarrollo, la nomenclatura de los mensajes de los commits y la depuración de responsabilidades a la hora de tener que realizar un *merge* manualmente.

Tal y como se vió al principio del proyecto, Django ha demostrado ser uno de los frameworks más potentes de los que existen ahora mismo sobre el papel. En cuanto a la experiencia de uso, hay que destacar su escasa curva de aprendizaje, que ha hecho mucho más llevadera la realización del trabajo. Sólo hay dos cosas que han costado más conseguir (ya que no están documentadas correctamente): encontrar información sobre cómo deshabilitar la protección contra cross site request forgery en los casos en que sea necesario hacer peticiones POST sin que éstas vengan de un formulario web y configurar correctamente el servidor para poder servir los ficheros estáticos (imágenes, css y ficheros javascript) a las distintas páginas web que se van mostrando.

Como se ha podido comprobar a lo largo del documento, se han cumplido exitosamente todos los objetivos propuestos inicialmente para el proyecto. Me gustaría destacar lo importantes que se prevee que sean los PaaS orientados a desarrollo de software en poco tiempo, según se explicó en la sección 2.3.1

4.3 Líneas futuras

Los próximos pasos a dar en la realización de este proyecto son la implementación de un módulo que permita la retrocompatibilidad con los servicios que ya tiene telefónica desplegados hasta ahora, a fin de evitar que éstos tengan que ser modificados para adaptarse al nuevo protocolo (lo que llevaría mucho tiempo, ya que el número de servicios desplegados es muy elevado), la observación del estado en que se encuentra el ecosistema de *addons* de Heroku (una revisión anual que permitirá

volver a evaluar los beneficios que aportan de cara a la relación de la facilidad de desarrollo que otorgan frente al precio que costarían en un entorno de producción'), y finalmente, dotar al sistema entero de alta disponibilidad, con lo que se evitarían los fallos que pueda generar la versión actual.

Cabe destacar que estos puntos los desarrollaré en el marco de una beca de formación en el laboratorio OIL UPM - Telefónica Digital durante mis estudios de Máster.

Apéndices

A Antiguo formato de SDR

Aquí se puede ver un ejemplo de SDR de los que se usan actualmente en telefónica (ya que la plataforma de *backoffice* que se ha desarrollado, y por consiguiente el nuevo formato de SDR tardarán en estar en el plan tecnológico de telefónica)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<fichero_consumos_variables>
<fecha_envio>2012-07-27T15:34:54.198Z</fecha_envio>
<consumos_variables total_registros= "4">
    <consumo_variable>
        <id>1</id>
        <contrato>003d000000kC2JHAA0</contrato>
        <servicio_comercial>0</servicio_comercial>
        <concepto_facturable>10052</concepto_facturable>
        <fecha_consumo>2012-07Z</fecha_consumo>
        <unidades>348.0</unidades>
        <posicion>1</posicion>
        <total_posiciones>1</total_posiciones>
    </consumo_variable>
</consumos_variables>
</fichero_consumos_variables>
```

	www.alrond.com		ab_5_1000 (II)			ab_100_10000 (II)	
		Req/sec	Time/req, ms	Transfer rate, Kb/s	Req/sec	Time/req, ms	Transfer rate, Kb/s
0	nginx	9440.20	0.106	2124.04	9602.59	0.104	2167.30
1	CodeIgniter (PHP)	92.57	10.803	31.84	91.16	10.969	31.42
2	Catalyst (Perl)	132.96	7.521	51.32	131.05	7.63	50.68
3	Django (Python-threaded)	767.13	1.304	289.21	752.96 *1)	1.328	284.85
4	Django (Python-threaded) + Psyco	969.06	1.032	365.33	1005.70	0.994	380.56
5	Django (Python-prefork)	694.18	1.441	261.70	590.82	1.693	223.27
6	Django (Python-prefork) + Psyco	894.24	1.118	338.92	682.86	1.464	258.05
7	RubyOnRails_1.1.6 (Ruby) * 2)	218.07	4.586	99.88	94.55	10.576	43.30
8	RubyOnRails_1.2.1 (Ruby) * 2)	59.99	16.670	27.47	43.82	22.822	20.07
9	Symfony (PHP)	28.35	35.272	29.71	28.17	35.497	29.55
10	TurboGears (Python-threaded)	164.50	6.079	54.12	163.83 *1)	6.104	53.92

Figura 8: Resultados del benchmark de frameworks web: Rendimiento

	www.alrond.com	Memory (start), KB		Memory (after I test), KB			Memory (after II test), KB		
		VSZ	RSS	VSZ	RSS	average %CPU	VSZ	RSS	average %CPU
0	nginx	3444+3*3800	592+3*1149	3444+3*3841	592+3*1189	0	3444+3*3841	592+3*1288	0
1	CodeIgniter (PHP)	6*8288	2988+5*1244	6*8288	2988+5*2764	5*4.48=22.4	6*8284	2988+5*2764	5*6.58=32.9
2	Catalyst (Perl)	6*18632	15704+5*14432	18632+5*20512	15704+5*17000	5*1.5=7.5	18636+5*20516	15704+5*17000	5*16.5=82.5
3	Django (Python-threaded)	50440	4364	94352	7236	1.6	54420	8844	20.4
4	Django (Python-threaded) + Psyco	87520	9336	117512	14980	1.4	96156	17804	12.6
5	Django (Python-prefork)	6*9488	4276+5*3624	9488+5*11156	4292+5*6192	* 3)	6*9488	4348+5*3640	* 3)
6	Django (Python-prefork) + Psyco	6*46560	9188+5*8537	46692+5*50417	9480+5*13883	* 3)	46688+5*46812	9516+5*8840	* 3)
7	RubyOnRails_1.1.6 (Ruby)	5*24608	5*21061	5*25284	5*21895	5*2.3=11.5	5*29420	5*26033	5*2.8=14
8	RubyOnRails_1.2.1 (Ruby)	5*24483	5*20948	5*25713	5*22322	5*1.92=9.6	5*28308	5*24921	5*16.06=80.3
9	Symfony (PHP)	6*8288	2992+5*1248	8288+5*8416	2992+5*3297	5*9.42=47.1	8284+5*8388	2988+5*3127	5*12.24=61.2
10	TurboGears (Python-threaded)	67492	14332	109988	16084	10	70428	17500	35.1

Figura 9: Resultados del benchmark de frameworks web: Memoria

B Comparativa de frameworks web

En enero de 2007 se enfrentaron los 6 frameworks más usados para desarrollo web a una serie de pruebas a fin de medir cual de ellos proporcionaba mejores características. Los resultados obtenidos fueron los siguientes:

En la figura 8 de la página 38 vemos los resultados de dos tests hechos usando Apache Benchmark. en la primera prueba (las tres primeras columnas de datos) se establece que se van a usar 5 hilos concurrentemente y que se van a recibir 1000 peticiones. En la segunda prueba (las tres últimas columnas) se realiza el mismo test pero con 100 hilos y 10000 peticiones. Cada una de las pruebas tiene, como se puede observar, 3 columnas. Son, por orden: Peticiones atendidas por segundo, tiempo por petición y velocidad de transferencia.

En la figura 9 de la página 38 vemos los resultados de memoria y uso de CPU

	www.alrond.com	Concurrent users	Transactions	Availability, %	Elapsed time, secs	Response time, secs	Transaction rate, trans/sec	Concurrency	Longest transaction	Shortest transaction	Memory after test, KB VSZ	RSS	average %CPU
0	nginx	50	5981	100	59.86	0.00	99.92	0.11	0.30	0.00			
		200	23600	100	60.04	0.00	393.07	0.48	0.21	0.00			
		300	35754	100	59.99	0.00	596.00	0.96	0.32	0.00	3448+3*3841	596+3*1199	3*0.13=0.4
1	CodeIgniter (PHP)	50	5001	100	60.08	0.07	83.24	5.74	0.77	0.01			
		200	6794	100	78.19	1.64	86.89	142.93	31.72	0.01			
		300	5956	99.65	78.28	2.99	76.09	227.29	28.90	0.01	6*8284	2988+5*2768	5*6.7=33.5
2	Catalyst (Perl)	50	5441	100	59.63	0.02	91.25	2.04	0.67	0.00			
		200	9714	100	76.82	0.78	126.45	98.43	9.82	0.00			
		300	10276	100	80.99	1.40	126.88	177.35	23.22	0.00	18716+5*20516	15704+5*17009	5*7.68=38.4
3	Django (Python-threaded)	50	5854	100	59.89	0.00	97.75	0.31	0.19	0.00			
		200	22456	100	60.02	0.01	374.14	3.91	0.69	0.00			
		300	29399	100	60.30	0.09	487.55	43.29	12.55	0.00	95504	8188	22.3
4	Django (Python-threaded) + Psyco	50	5991	100	59.81	0.00	100.17	0.34	0.21	0.00			
		200	23452	100	60.18	0.01	389.70	2.59	0.44	0.00			
		300	32253	100	60.93	0.02	529.35	12.01	8.44	0.00	134796	15700	18.5
5	Django (Python-prefork)	50	6035	100	60.84	0.00	99.19	0.35	0.37	0.00			
		200	21674	100	60.65	0.05	357.36	17.00	3.67	0.00			
		300	22489	100	64.15	0.35	350.57	121.88	9.34	0.00	6*9480	4340+5*3629	* 2)
6	Django (Python-prefork) + Psyco	50	5891	100	60.22	0.01	97.82	0.59	0.70	0.00			
		200	22130	100	59.99	0.03	368.89	11.23	5.30	0.00			
		300	23893	100	60.92	0.24	392.20	94.11	25.99	0.00	6*46688	9564+5*8832	* 2)
7	RubyOnRails_1.1.6 (Ruby)	50	5737	100	60.07	0.01	95.51	1.20	0.64	0.00			
		200	8291	99.99	65.50	0.98	126.58	123.97	23.69	0.00			
		300	7987	99.54	64.53	1.20	123.77	148.93	29.83	0.00	5*33458	5*30072	5*9.86=49.3
8	RubyOnRails_1.2.1 (Ruby)	50	5751	100	60.11	0.02	95.67	1.56	1.16	0.00			
		200	8525	99.99	70.64	1.05	120.68	127.18	25.98	0.00			
		300	6698	99.58	73.51	2.30	91.12	209.29	29.94	0.00	5*36593	5*33207	5*7.44=37.2
9	Symfony (PHP)	50	1701	100	60.37	1.25	28.18	35.28	2.42	0.20			
		200	1675	99.64	59.90	5.10	27.96	142.51	29.88	0.03			
		300	2087	97.34	75.35	4.36	27.70	120.88	29.85	0.03	8288+5*8396	2992+5*3139	5*8.24=41.2
10	TurboGears (Python-threaded)	50	5632	100	60.33	0.01	93.35	1.32	0.32	0.00			
		200	9351	100	61.29	0.64	152.57	97.35	25.54	0.00			
		300	12061	100	79.32	0.88	152.05	134.12	30.32	0.00	111164	17532	54.0

Figura 10: Resultados del benchmark de frameworks web: Escalabilidad

obtenidos con los tests. hay 3 bloques: datos antes de empezar, datos tras acabar el primer test y datos tras acabar el segundo test. Cabe destacar que en las filas en que se usa Python con prefork no se pueden obtener datos fiables, ya que ante una carga grande los procesos se reinician.

En la figura 10 de la página 39 se muestran los resultados para otro de los test que realizó el mismo autor que el anterior. Esta vez se usó la aplicación siege, que coloca tu aplicación web bajo una sobrecarga de peticiones a fin de comprobar la fiabilidad del servidor web, si pierde datos o si hay momentos en que no está disponible. Se llevaron a cabo 3 pruebas con esta herramienta (todas de una duración de 1 minuto): con 50 usuarios simultáneos, con 100 usuarios simultáneos y con 300 usuarios simultóneos.

Como se puede comprobar de los datos arrojados por dicho estudio, no hay duda de que Django es el framework web con mejores prestaciones.

Bibliografía

[1] DocumentCloud. Backbone.js, 2012.

[2] Alrond. The performance test of 6 leading frameworks. *http://www.alrond.com/*, 2007.

[3] Falvey Ryan. Regulation of the cloud in india. *Journal of Internet Law*, 15(4), 2011.

[4] Simson L. Garfinkel. The cloud imperative. *technologyreview*, 2011.

[5] Peter Mell and Timothy Grance. The nist definition of cloud computing. 2011.

[6] Alexander Linden and Jackie Fenn. Understanding gartner's hype cycles, 2003.

[7] George Gilder. The information factories. *Wired magazine*, 14(10):1–5, 2006.

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Fri Feb 14 19:14:24 CET 2014
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)